Theoretical Assessment: DevOps Strategy

The app is simple, it takes one image in a bucket and resizes it before uploading it to a second one. There was no specific requirement beside that so I decided to keep it simple and not to include extra stuff in the code to return this test in a short time. I will try to explain the addition I would have added here.

1. **Observability:**

○ **How would you implement comprehensive observability for this thumbnail generation service? What metrics, logs, and traces would you collect? What tools would you use to analyze and visualize this data?**

The main metric we can monitor for this app is the lambda error rate, that would indicate any big issue related to the code. In addition to that, monitoring the average execution time and the number of invocation of the lambda can be important as lambda are billed based on execution time. In addition to monitoring the lambda, there are some metrics we can use for the S3 too, like the difference in the number of objects between the two buckets.

As for the logs, we can monitor the errors the lambda catches when it can't resize an image or when the format is not recognized to maybe implement the necessary changes to the code.

For the tooling used to analyze and visualize those metrics i would use cloudwatch directly as the app is simple, but having tools like Grafana for the metrics and Kibana for the logs can make it easier for developers to access those data.

2. **Security:**

○ **Outline a security strategy for this service. How would you secure the S3 buckets, the Lambda function, and the data in transit? How would you protect against common vulnerabilities?**

For the S3 bucket we can implement the principle of least privilege, and only allow access to entities who need it. By default AWS creates a bucket without any public access, i would keep that in place and only grant read and write to the lambda depending on the bucket. Server side encryption is also enabled by default so no change to do here. One additional protection would be to implement versioning to avoid accidental deletion.

For the Lambda, we need to keep the access right to only the necessary things (so access to the two buckets) and avoid giving more rights than that. We do not need to use a VPC for the lambda as we are accessing resources that are not compatible with that. Regarding data in transit, boto is just a wrapper of the aws api and uses https by default. Validating the s3 message sent to the lambda can be interesting too, but as that's our only trigger it's not likely to be an issue.

Finally to protect against common vulnerability we need to keep the library used by the lambda up to date (So boto / botocore / Pillow)

3. **Monitoring:**

○ **Describe how you would set up proactive monitoring to detect and alert on issues like failed thumbnail generation, latency spikes, or resource exhaustion. What specific thresholds and alerts would you configure?**

I would put alerts using cloudwatch metrics on error rate, and spike in average latency. Additionally I would monitor the number of concurrent executions to make sure lambda is not throttling.
For this app I think we can set low thresholds as the code is simple, maybe a 1% error rate over 15 minutes, or a 1 second p99 execution duration. Additionally we can compare the number of objects on the two buckets to get an idea of the all time error rate.

An advantage of using tools like Datadog/Cloudwatch anomaly detection in addition to just classical metrics is the ability to detect in advance changes in the behavior of the app by monitoring the metrics as well as the logs, it can quickly detect changes in patterns and alert us before it triggers a threshold alert.

4. **CI/CD:**

○ **Design a CI/CD pipeline for this service. Explain how you would automate the build, test, and deployment process. How would you ensure that changes are deployed safely and reliably?**

As the app is simple and uses only serverless products I would handle the code of the lambda and the code of the stack defined with the aws cdk at the same time. Everything versioned with Git.
For the pipeline we can use Github action or AWS CodePipeline (and many more). As the app is serverless and does not cost anything if we do not use it, having a test environment is "free". So to keep it simple i would implement the following:

- Two branches "main" and "staging" and as many feature branches as needed, the main branch is protected and only allows Pull requests with reviews.
- On every pull request I would run the two test files (one for the lambda one for the stack) and maybe a lint step. A nice thing to have too also could be an automatic change detection on the stack publish to the PR with the associated cost

- When a pull request is merged on staging, we run the test and deploy the app to the corresponding environment.

- For the main branch (prod env) we could either have the deploy run automatically like staging, or have a manual trigger to avoid any unwanted deploy to the production stack.

The advantage with this app as we do not use any database is that it's quite easy to rollback versions, and we do not really need any fancy deployment strategy like canaries, blue/green deployment or feature flags.

5. **Scalability and Cost Optimization:**

○ **Discuss how you would design this solution for scalability. How would you handle increased traffic and load? What cost optimization strategies would you implement?**

For the scaling we can skip S3 as we do not control anything and it virtually scales indefinitely. However for the lambda we need to be careful with the setup we want. The concurrency is the main setting we can tweak to make sure the service scales. Maybe we can think of using Lambda Provisioned Concurrency at some point for more consistency too.
https://quintagroup.com/blog/aws-lambda-provisioned-concurrency

For the cost optimization part, we can set up life-cycle rules on s3 to change the storage class of the object stored based on their age/access. And adapt the lambda memory/cpu rules based on what we know the lambda needs.

One of the main changes I would make to this app is adding a "Buffer" between S3 event and the Lambda. By Using SQS for example we can implement a proper retry mechanism and even create a dead-letter queue to avoid having the retry implemented in the lambda and to be able to properly manage and replay incidents. Here is a nice article on why retry in the code is not necessarily a good idea:
https://serverlessdna.com/strands/observability/when-serverless-goes-wrong
(Note that the DLQ is also a feature accessible directly in lambda)

Another approach to this problem would be to use Cloudfront and lambda@Edge to handle the resizing. Lambda@Edge are not as cost effective but by using cloudfront we could avoid having to recreate a new thumbnail if we already have it in cache, and we could also dynamically ask for any size directly on the request to allow for more flexibility. One big advantage also is the ability to reduce the cost of traffic going out of S3, it's more cost effective to use cloudfront and we can negotiate private pricing with less difficulty than with s3.

Example implemented at my current job:
https://assets.dev.emerald.sekai.gg/arc_banner.png
https://assets.dev.emerald.sekai.gg/arc_banner.png?height=100&width=50